Lecture 5
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Multithreading
Computing with Graphical Processing Units
Announcements

• Get ready for CUDA!
  ➢ Run the incrArray code example on Sorken and Stampede

• A2: GPU programming matrix multiplication
Today’s lecture

• Programming with OpenMP (continued from last lecture)
• Introduction to GPUs - architecture
Progress with Assignment #1

8.14 GFlops

$R_{\infty} = 4 \times 2.33 = 9.32 \text{ Gflops}$

$\approx 87\% \text{ of peak}$
#include <emmintrin.h>

void square_dgemm (int N, double* A, double* B, double* C)
{
    __m128d c1 = _mm_loadu_pd( C+0*N);  // load unaligned block in C
    __m128d c2 = _mm_loadu_pd( C+1*N);
    for( int i = 0; i < 2; i++ ){
        __m128d a1 = _mm_load1_pd( A+i+0*N);  // load i-th column of A (A0i,A0i)
        __m128d a2 = _mm_load1_pd( A+i+1*N);
        __m128d b = _mm_load_pd( B+i*N);  // load aligned i-th row of B
        c1 = _mm_add_pd( c1, _mm_mul_pd( a1, b ) );  // rank-1 update
        c2 = _mm_add_pd( c2, _mm_mul_pd( a2, b ) );
    }
    _mm_storeu_pd( C+0*N, c1 );  // store unaligned block in C
    _mm_storeu_pd( C+1*N, c2 );
}

C00_C01 += A00_A00 * B00_B01
C10_C11 += A10_A10 * B00_B01
C00_C01 += A01_A01 * B10_B11
C10_C11 += A11_A11 * B10_B11
Parallelizing a loop nest with OpenMP

- We parallelize the outer loop index, indicating shared and private (local) variables
- Some implementations can parallelize inner loops

```c
#pragma omp parallel private(i) shared(n)
#pragma omp for
for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
        u\_new[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])/4
    }
```

- Generated code
  - An implicit barrier after the loop

```c
mymin = 1 + ($TID * n/nproc), mymax = mymin + n/nproc - 1
for(i=mymin; i < mymax; i++)
    for(j=0; j < n; j++)
        u\_new[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])/4
    Barrier();
```
Variable scoping

- Any variables declared outside a parallel region are shared by all threads
- Variables declared inside the region are private
- Used `shared` and `private` declarations to override the defaults

```c
double c = 1 / 6.0, h = 1.0, c2 = h * h;
double ***c = ...;
for (it= 0; it<nIters; it++) {
    #pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
    #pragma omp for
    for (int i=1; i<=nx; i++)
        for (int j=1; j<=ny; j++)
            for (int k=1; k<=nz+1; k++)
                Un[i][j][k] = c* (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] +
                                  U[i][j][k-1] + U[i][j][k+1] - c2*b[i-1][j-1][k-1]);
    Swap U ↔ Un;
}
```
An application: Matrix Vector Multiplication

- The matrix is 2D array, but is memory is 1 dimensional
- We will assume *row major* order
  (Fortran uses column major)

\[
\begin{array}{cccc}
  a_{00} & a_{01} & \cdots & a_{0,n-1} \\
  a_{10} & a_{11} & \cdots & a_{1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

\[
\begin{array}{c}
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\end{array} =
\begin{array}{c}
y_0 \\
y_1 \\
\vdots \\
y_{m-1} \\
\end{array}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Initialization

• Allocate and initialize storage outside a parallel region
• In some applications, we initialize in parallel in order to optimize for NUMA

```c
double **A;
A = new double [N*N];
assert(A);

for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i*N+j] = 1.0 / (double) (i+j-1);
```
double *A, *x, *y;
// Start timer
   double t0 = -getTime();

#pragma omp parallel shared(A,x,N)
   for (int k = 0; k<reps; k++)
#pragma omp for
   for (i=0; i<N; i++)
       y[i] = 0.0;
   for (j=0; j<N; j++)
       y[i] += A[i*n+j] * x[j];

// Take time
   t0 += getTime();
Reductions in OpenMP

• What if we need to reduce a set of values down to a single value?
• We need to tell OpenMP to ensure atomicity when updating sum, to avoid the *race condition*

```c
#pragma omp parallel reduction(+:sum)
  for (int i=0; i < N-1; i++)
    gsum += f(x[i]);

i0 = $TID*n/$nthreads, i1 = i0 + n/$nthreads;
for (i=i0, localSum=0; i < i1; i++)
  localSum += f(x[i]);
gsum.atomicAdd(localSum);
```
Race conditions

- Sometimes a program may exhibit non-deterministic behavior: results are not correct
- This can happen when there is a conflict when updating a shared quantity: a data race
  - Arises when there is at least one writer on shared data
  - The timing of accesses to shared data can affect the outcome
Under the hood of a data race

• Consider this statement, assume \( x == 0 \)
  \[
x = x + 1;
\]

• Generated code
  \[
  \begin{align*}
    r1 &\leftarrow (x) \\
    r1 &\leftarrow r1 + #1 \\
    r1 &\rightarrow (x)
  \end{align*}
  \]

• Possible interleaving with two threads
  \[
  \begin{array}{c}
    P1 \\
  \end{array}
  \begin{array}{c}
    P2 \\
  \end{array}
  \]
  \[
  \begin{align*}
    r1 &\leftarrow x \\
    r1 &\leftarrow r1 + #1 \\
    x &\leftarrow r1
  \end{align*}
  \]
  \[
  \begin{align*}
    r1 &\leftarrow x \\
    r1 &\leftarrow r1 + #1 \\
    x &\leftarrow r1
  \end{align*}
  \]
  \[
  \begin{align*}
    r1(P1) &\text{ gets } 0 \\
    r2(P2) &\text{ also gets } 0 \\
    r1(P1) &\text{ set to } 1 \\
    r1(P1) &\text{ set to } 1 \\
    P1 &\text{ writes its R1} \\
    P2 &\text{ writes its R1}
  \end{align*}
  \]
Avoiding data races

• Usually we want to avoid non-determinism
• If we compute with the same inputs we want to obtain the same results
• Not necessarily true for operations that have side effects (global variables, I/O and random number generators)
• We need to take steps to avoid race conditions through appropriate program synchronization
  ♦ Atomic variables
  ♦ Critical sections
  ♦ Barriers
• Very difficult to detect atomicity errors
• Failing to employ synchronization in multithreaded code doesn’t guarantee safety violations, it just allows them!
Barriers in OpenMP

• Even if we avoid the race condition through an atomic update, we still need the barrier
• Why?

```c
#pragma omp parallel reduction(+:sum)
    for (int i=0; i < N-1; i++)
        gsum += f(x[i]);
Print gsum

i0 = $TID*n/nthreads, i1 = i0 + n/nthreads;
for (i=i0, localSum=0; i < i1; i++)
    localSum += f(x[i]);
gsum.atomicAdd(localSum);
Print gsum
```
Incorrect behavior

- OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code
  
  ```c
  int* fib = new int[N];
  fib[0] = fib[1] = 1;
  #pragma omp parallel for num_threads(2)
  for (i=2; i<N; i++)
    fib[i] = fib[i-1] + fib[i-2];
  ```

- OpenMP may warn you when it is doing something unsafe, but not always
Recall processor design trends

• No longer possible to use growing population of transistors to boost single processor performance
  ‣ Can no longer increase the clock speed, since power consumption grows with the clock frequency
  ‣ Instead, we replicate the cores

• An opportunity: Specialize the processing core
  ‣ Simplified design, pack more onto the chip
  ‣ Boost performance
  ‣ Reduce power

• Simplified core
  ‣ Remove architectural enhancements like branch caches
  ‣ Constrain memory access and control flow
  ‣ Partially expose the memory hierarchy
Computing with many cores

• We’ll look at one member of the family—Graphical Processing Units—made by one manufacturer—NVIDIA

• Simplified core, replicated on a grand scale: 1000s
  ‣ Remove power hungry features of modern processors
  ‣ No branch caches
  ‣ Constrain memory access (aligned) and control flow (conditionals)
  ‣ Explicit on chip-data motion

• An opportunity: specialize the processing core
  ‣ Simplified design, pack more onto the chip
  ‣ Boost performance
  ‣ Reduce power
Heterogeneous processing with Graphical Processing Units

- Specialized *many-core* processor (the device) controlled by a conventional processor (the host)
- Explicit data motion
  - between *host* and *device*
  - inside the device
What are GPUs good at?

• Processing long vectors on thousands of specialized cores
• Executing 1000s of threads to hide data motion
• Some regularity involving memory accesses and control flow
Stampede’s NVIDIA Tesla Kepler K20m (GK110)

- Hierarchically organized clusters of streaming multiprocessors
  - 13 streaming processors @ 705 MHz
    (down from 1.296 GHz on GeForce 280)
  - Peak performance: 1.17 Tflops/s Double Precision, fused multiply/add
- SIMT parallelism
- 5 GB “device” memory (frame buffer) @ 208 GB/s
  www.techpowerup.com/gpudb/2029/tesla-k20m.html
Sorken’s K80 GPU (GK210)

- 2 GPUs, compute capability 3.7 (3.5 for K20) each with
  - 13 streaming processors @ 745 MHz
  - Peak performance: 1.17 Tflops/s Double Precision, fused multiply/add
  - 128K 32 bit registers, 64K for 3.5 capability
- SIMT parallelism
- 11.25 GB device memory @ 240 GB/s
- See [www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu](http://www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu)
Overview of Kepler GK110

Multicore CPU

Hyper-Q Execution Manager

Warp Scheduler
Dispatch
Single Precision
Double Precision

SMX14

User Selectable Hardware/Software Cache

Level 2 Hardware Cache

Stream Optimized Device Memory

Latency Optimized Host Memory

Control

DMA

©2013 The Portland Group, Inc.

10/7/15 Scott B. Baden / CSE 260, UCSD / Fall '15
SMX Streaming processor

- AKA vector unit
- Stampede’s K20s (GK110 GPU) have 13 SMXs (2496 cores), so do K80s
- Each vector unit
  - 192 SP cores, 64 DP cores, 32 SFUs, 32 Load/Store units
  - Each scalar core: fused multiply adder, truncates intermediate result
  - 64KB (K80:112K) on-chip memory configurable as scratchpda memory + L1 cache
  - 64K x 32-bit registers (256KB) up to 255/thread
  - 1 FMA /cycle = 2 flops / cyc / DP core * 64 DP/SMX * 13 SMX = 1664 flops/cyc @0.7006 Ghz = 1.165 TFLOPS
Kepler’s Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Can partition the on-chip Shared memory L,1$ cache
  $\{\frac{3}{4} + \frac{1}{4}\}$
  $\{\frac{3}{4} + \frac{1}{4}\}$
  $\{\frac{1}{2} + \frac{1}{2}\}$
- L2 Cache (1.5 MB)
CUDA

- Programming environment with extensions to C
- Under control of the *host*, invoke sequences of multithreaded kernels on the *device* (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads $\text{Grid} \supset \text{Block} \supset \text{Thread}$
- Hardware dispatches blocks to cores, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation

KernelA$<<<2,3>,<3,5>>>()$
Thread block execution

- **Thread Blocks**
  - Unit of workload assignment
  - Each thread has its own set of registers
  - All have access to a fast on-chip *shared memory*
  - Synchronization among all threads in a block only
  - Threads in different blocks communicate via slow global memory
  - Processor groups threads into *warps* of 32 threads
- **SIMT parallelism:** all threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization

```
KernelA<<<2,3>,<3,5>>>()
```

David Kirk/NVIDIA & Wen-mei Hwu/UIUC

Scott B. Baden / CSE 260, UCSD / Fall '15
Warp Scheduling

• Threads assigned to an SMX in units of a thread block, multiple blocks
• Each block divided into warps of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a Warp execute the same instruction, branches serialize execution
• Multiple warps simultaneously active, hiding data transfer delays
• All registers in all the warps are available, 0 overhead scheduling
• Hardware is free to assign blocks to any SMX
• There are 4 warp schedulers/SMX
Coding example – Increment Array

Serial Code

```c
void incrementArrayOnHost(float *a, int N){
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f; }
```

CUDA

```
// Programmer determines the mapping of virtual thread IDs
// to global memory locations
#include <cuda.h>
__global__ void incrementOnDevice(float *a, int N) 
{ // Each thread uniquely specified by block & thread ID
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+1.f;
}
incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```
Managing memory

- Data must be allocated on the device and host and moved between the two explicitly

```c
float *a_h, *b_h;          // pointers to host memory
float *a_d;                // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i;  // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N,
            cudaMemcpyHostToDevice);
```
Computing and returning result

```c
int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
    cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

// check results
    for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
    free(a_h); free(b_h);
    cudaFree(a_d);
```
Experiments - increment benchmark

• Total time: timing taken from the host, includes copying data to the device
• Device only: time taken on device only

N = 8388480, block size = 128, times in milliseconds, cseclass02
Reps = 10 100 1000 10⁴ 10⁵

<table>
<thead>
<tr>
<th>Reps</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device</td>
<td>Device</td>
<td>Device</td>
<td>Device</td>
<td>Device</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>10</td>
<td>3.3</td>
<td>36</td>
<td>358</td>
<td>3.58s</td>
<td>35.8s</td>
</tr>
<tr>
<td>100</td>
<td>71.</td>
<td>102</td>
<td>429</td>
<td>3.64s</td>
<td>35.9s</td>
</tr>
<tr>
<td>1000</td>
<td>92.</td>
<td>730</td>
<td>7.06s</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>10⁴</td>
<td>6.8</td>
<td>52</td>
<td>500</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>10⁵</td>
<td>6.4s</td>
<td>23.9s</td>
<td>200s</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Device time
Kernel launch + data xfer
Host
Sine function (Host)
What is the cost of moving the data and launching the kernel?

A. About 6.8 ms \((71-3)/10\)
B. About 0.66 ms \((102-36)/100\)
C. About 0.071 ms \((429-358)/1000\)
D. About 68 ms \(71-3\)

N = 8 M block size = 128, times in \textit{milliseconds}

<table>
<thead>
<tr>
<th>Reps</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>(10^4)</th>
<th>(10^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device time (increment)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>36</td>
<td>358</td>
<td>3.58s</td>
<td>35.8s</td>
<td></td>
</tr>
<tr>
<td>71.</td>
<td>102</td>
<td>429</td>
<td>3.64s</td>
<td>35.9s</td>
<td></td>
</tr>
<tr>
<td>Kernel launch + data xfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scott B. Baden / CSE 260, UCSD / Fall '15